




VIRTUAL COMPUTER HAVING JIT COMPILER

Patent number: JP2003140909
Publication date: 2003-05-16
Inventor: SHIMURA HIROYA
Applicant: FUJITSU LTD
Classification:
 - international: G06F9/45
 - european:
Application number: JP20010341577 20011107
Priority number(s):

Also published as:

 EP1445695 (A1)
 WO03040918 (A1)
 US2004210865 (A1)

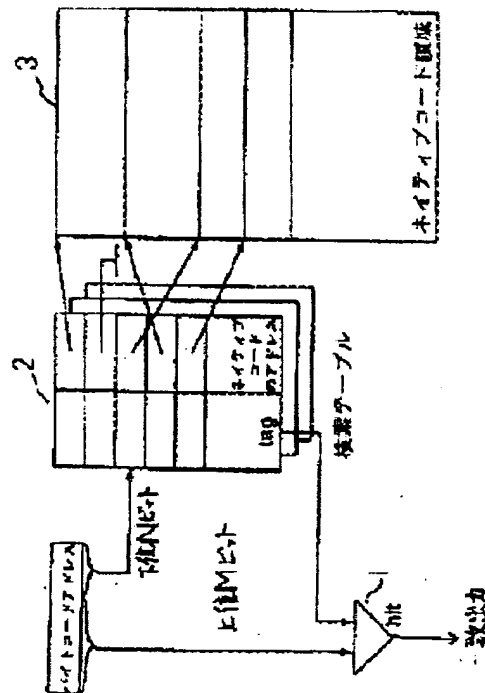
Report a data error here

Abstract of JP2003140909

PROBLEM TO BE SOLVED: To improve responsiveness by increasing speed of processing within a limited memory capacity in relation to a virtual computer having a JIT compiler.

SOLUTION: This virtual computer comprises a function of interpreting and executing a bite code of the virtual computer with software, and a code cache as a storing region of a native code having the limited capacity on a memory. The computer also comprises the JIT compiler that compiles the bite code to a native code capable of being directly executed by the virtual computer in executing the bite code, stores the native code in the code cache 3, and then executes the native code. The computer also comprises a retrieving table 2 for retrieving information used for determining whether the native code is compiled and an address of the compiled native code based on the address of the bite code.

例1の検索テーブル例



(19)日本国特許庁 (J P)

(12) 公 開 特 許 公 報 (A)

(11)特許出願公開番号

特開2003-140909

(P2003-140909A)

(43)公開日 平成15年5月16日(2003.5.16)

(51)Int.Cl.⁷

G 0 6 F 9/45

識別記号

F I

C 0 6 F 9/44

データベース(参考)

3 2 0 C 5 B 0 8 1

審査請求 未請求 請求項の数5 O L (全 17 頁)

(21)出願番号 特願2001-341577(P2001-341577)

(22)出願日 平成13年11月7日(2001.11.7)

(71)出願人 000003223

富士通株式会社

神奈川県川崎市中原区上小田中4丁目1番
1号

(72)発明者 志村 浩也

神奈川県川崎市中原区上小田中4丁目1番
1号 富士通株式会社内

(74)代理人 100096530

弁理士 今村 辰夫 (外2名)

Fターム(参考) 5B081 AA09 CC41 DD01

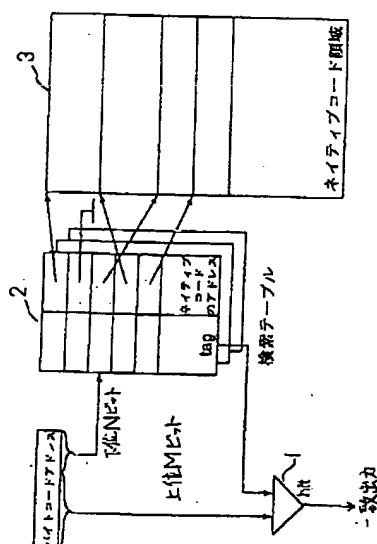
(54)【発明の名称】 J I Tコンパイラを備えた仮想計算機

(57)【要約】

【課題】本発明はJ I Tコンパイラを備えた仮想計算機に関し、限られたメモリ容量で処理の高速化を図り、応答性能の向上を図る。

【解決手段】仮想計算機のバイトコードをソフトウェアで解釈・実行する機能を有し、メモリ上に容量の制限されたネイティブコードの格納領域としてコードキャッシュを持ち、バイトコードを実行する際にバイトコードを仮想計算機が直接実行できるネイティブコードにコンパイルしてコードキャッシュ3に格納した後、そのネイティブコードを実行するJ I Tコンパイラを備え、バイトコードのアドレスから、コンパイルされているか否かを判断するための情報と、コンパイルされたネイティブコードのアドレスを検索する検索テーブル2を備えた。

例1の検索テーブル例



理である。

【0010】前記インタプリタの処理速度が遅いことを解決する方法として、JIT (Just-IN-Compiler) コンパイラを使う手法が一般に使用されていた。このJITコンパイラはバイトコードをネイティブコードにコンパイルする。これにより、アプリケーションは直接そのマシンのプロセッサで実行できるようになり、高速な実行が可能になる。

【0011】一般に、JITコンパイラは、ダウンロードしたクラス、又は実行を開始したメソッドという単位でコンパイルを行う。すなわち、メソッドのバイトコード全体をコンパイルする。また、コンパイルしたネイティブコードは、メモリ（プログラム上の格納領域）上に格納され、再び実行を行う時は再利用され、1度目の実行時には、コンパイルのオーバーヘッドがあるが、2度目からはこのオーバーヘッド無しに高速に実行できる。具体的には次のようになる。

【0012】まず、バイトコードの命令フェッチを行い（S11）、フェッチした命令をデコードする（S12）。そして、デコードした結果が実行であれば、その命令を実行し（S13、S14）、S11の処理へ移行する。また、デコードした結果が分岐であれば、分岐処理を行い（S15）、S11の処理へ移行する。更に、デコードした結果がメソッド呼び出しであれば、メソッド呼び出しを行い（S16）、コンパイル済みか否かを判断する（S17）。

【0013】その結果、コンパイル済みでなければ、コンパイルすべきか否かを判断し（S18）、コンパイルすべきでなければ、S11の処理へ移行するが、コンパイルすべきであれば、メソッドで呼び出したコードをJITコンパイラによりコンパイルしてネイティブコードを生成し、ネイティブコードの格納領域に格納し（S19）、ネイティブコードを実行する（S20）。ネイティブコードの実行中、メソッド呼び出しがあれば、S16の処理へ移行し、メソッドの処理が終了すれば、S11へ移行する。また、S17の処理において、コンパイル済みであれば、S20の処理へ移行する。以降、同様にして処理を行う。

【0014】なお、前記バイトコード (bytecode) とは、Javaで開発したソフトのバイナリ表現形式として米サン・マイクロシステムズが定めたJava仮想マシン（仮想計算機）のマシン・コードのことを言う。また、バイトコードをマシンが直接実行できるネイティブコードへ変換するソフトを「JIT (Just-in-time) コンパイラ」と呼ぶ。

【0015】また、「メソッド」 (Method) とは、オブジェクト指向プログラミングにおいて、オブジェクトの実行する操作の方法を記述したプログラム（サブルーチンと同じ意味）のことを言う。

【0016】

【発明が解決しようとする課題】前記のような従来のものにおいては、次のような課題があった。

【0017】すなわち、携帯電話機やPDA等の携帯型無線通信機器では使用できるメモリ容量が限られており、コンパイルしたコードを全てメモリ上に格納することはできない。また、前記携帯型無線通信機器でのソフトウェアの実行は、ゲームなどの対話型のアプリケーションが多く、応答性を高めるためには、コンパイルにあまり時間をかけることはできない。

【0018】特に、前記携帯型無線通信機器に搭載されたプロセッサでは処理能力が比較的低いため、大きなプログラムをコンパイルすると、実行中のプログラムが一瞬止まって見えることもある。また、一度しか実行されないようなバイトコードの場合、コンパイルする時間のオーバーヘッドのため、インタプリタで実行するよりも遅くなる。

【0019】本発明は、このような従来の課題を解決し、限られたメモリ容量で処理の高速化を図り、応答性能の向上を図ることを目的とする。

【0020】

【課題を解決するための手段】本発明は前記の目的を達成するため、次のように構成した。

【0021】(1)：仮想計算機のバイトコードをソフトウェアで解釈・実行する機能を有し、メモリ上に、容量の制限されたネイティブコードの格納領域としてコードキャッシュを持ち、前記バイトコードを実行する際に、JITコンパイラによりバイトコードを仮想計算機が直接実行できるネイティブコードにコンパイルして前記コードキャッシュに格納した後、そのネイティブコードを実行するJITコンパイラを備えた仮想計算機において、前記バイトコードのアドレスから、コンパイルされているか否かを判断するための情報と、コンパイルされたネイティブコードのアドレスを検索する検索テーブルを備えていることを特徴とする。

【0022】(2)：前記(1)の仮想計算機において、前記検索テーブル上、又は前記コードキャッシュ上に、命令の実行回数を計数するための実行カウンタを設け、バイトコードを実行する際に、前記実行カウンタの値を調べ、予め決めた特定回数より小さい場合はインタプリタで実行し、前記特定回数以上の場合は、JITコンパイラによりバイトコードをコンパイルしてから実行する選択処理手段を備えていることを特徴とする。

【0023】(3)：前記(1)の仮想計算機において、前記検索テーブルに、1度目の実行時にバイトコードのアドレスだけ登録し、ネイティブコードのアドレスの登録は特別な値、又は数字の0を登録しておく検索テーブル情報登録手段と、1度目はインタプリタで実行し、2度目はJITコンパイルによりコンパイルして実行する実行制御手段を備えていることを特徴とする。

【0024】(4)：前記(1)の仮想計算機において、コ

パイラ（以下、単に「コンパイラ」とも記す）を備えた仮想計算機に関するものであり、バイトコードのアドレスから、コンパイルされているか否かを判断するための情報（後述する）と、コンパイルされたネイティブコードのアドレスを検索する検索テーブルを備えている。

【0040】そして、コンパイルを行う時にネイティブコードの格納領域に空きがなければ、既に格納済みのコンパイルコードを破棄して前記格納領域に空き領域を作り、検索テーブルを更新するものである。

【0041】②：フローチャートによる処理（その1）の説明

図1は例1の処理フローチャート（その1）である。以下、図1に基づいて、例1の処理（その1）を説明する。なお、S31～S40は各処理ステップを示す。この処理では、全てのバイトコードを対象としてコンパイルされているかどうかをチェックする例であり、仮想計算機上の処理である。

【0042】まず、全てのバイトコードに対して、検索テーブル上のタグ（tag）情報を利用して（詳細は後述する）コンパイル済みか否かを判断し（S31）、コンパイル済みでなければ、コンパイルすべきか否かを判断し（S32）、コンパイルすべきでないと判断した場合には、バイトコード命令のフェッチを行い（S33）、フェッチされた命令のデコードを行う（S34）。

【0043】その結果、実行であれば（S35、S36）その命令を実行し、S31の処理へ移行する。また、分岐であれば（S37）分岐処理を行い、S31の処理へ移行する。更に、メソッド呼び出しであれば（S38）、メソッド呼び出しを行い、S31の処理へ移行する。

【0044】また、前記S32の処理で、コンパイルすべきであると判断した場合には、部分的にコンパイルし、ネイティブコードを生成してネイティブコード格納領域へ格納し（S39）、ネイティブコードを実行し（S40）、S31の処理へ移行する。また、S31の処理で、コンパイル済みであれば、S40の処理へ移行する。前記処理において、S31～S38はインタプリタの処理であり、S39、S40がJITコンパイラによる処理である。

【0045】③：フローチャートによる処理（その2）の説明

図2は例1の処理フローチャート（その2）である。以下、図2に基づいて、例1の処理（その2）を説明する。なお、S41～S50は各処理ステップを示す。

【0046】この処理では、分岐、メソッド呼び出しなどの実行制御が途切れる地点で、コンパイルのチェックを行う。一般のJITコンパイラとの違いとしては、コンパイルをメソッド単位で行わず、以下に説明する例5、例6、例7、例8等の条件で、コンパイルを中断す

ることが特徴である。

【0047】まず、バイトコード命令のフェッチを行い（S41）、命令デコードを行う（S42）。その結果、実行であれば（S43、S44）、その命令を実行し、S41の処理へ移行する。また、分岐であれば（S45）、分岐処理を行い（S45）、メソッド呼び出しであれば（S46）メソッド呼び出しを行う。

【0048】そして、分岐又はメソッド呼び出しを行った場合、コンパイル済みか否かを判断し（S47）、コンパイル済みでなければ、コンパイルすべきか否かを判断する（S48）。その結果、コンパイルすべきでないと判断した場合には、S41の処理へ移行する。

【0049】また、コンパイルすべきであると判断した場合は、途中までコンパイルを行い、ネイティブコードを生成しネイティブコード格納領域へ格納する（S49）。そして、ネイティブコードを実行し（S50）、S47の処理、又はS46の処理へ移行する。

【0050】④：検索テーブルの説明

図3は例1の検索テーブル例である。前記のようにコンパイル済みか否かを調べて判断するために、検索テーブルを使用するが、その検索テーブルの1例を図3に示す。図3において、1はコンパレータ、2は検索テーブル、3はコードキャッシュである。

【0051】この検索テーブル2は、タグ（tag）と、ネイティブコードのアドレス（コードキャッシュ3内のネイティブコード領域のアドレス）との対応情報が登録できるようになっており、ネイティブコードのアドレスからコードキャッシュ3のネイティブコード領域のデータが検索できるようになっている。

【0052】また、検索テーブル2は、バイトコードアドレス（上位Mビット+下位Nビットとする）の下位Nビットを利用してタグ（tag）を検索することで、ネイティブコードのアドレスが検索できるようになっている。そして、バイトコードアドレスの上位Mビットをコンパレータ1に入力すると共に、検索テーブルのタグ（tag）情報をコンパレータ1に入力し、両者が一致するか否かの比較を自動的に行うようになっている。

【0053】前記比較の結果、両者が一致（hit）した場合、コンパレータ1から一致出力が出され、この一致出力信号によりコンパイル済みと判断できるようになっている。

【0054】⑤：例1の特徴

前記図3の検索テーブルを検索することでコンパイルされているか否かを判断したり、コンパイルされている場合に、該当するネイティブコードの格納アドレス（コードキャッシュ3内のネイティブコード領域のアドレス）を検索することが容易になる。

【0055】その結果、バイトコードをネイティブコードにコンパイルすることで、アプリケーションプログラムの実行時間を速くすることが可能になる。また、従来

【0069】また、S81の処理で、テーブル登録済みであると判断した場合（2度目）は、JITコンパイラによる処理を行う。この場合、ネイティブコードアドレスへジャンプし（S91）、エントリBからJITコンパイラによるコンパイルを行い（S92）、ネイティブコードアドレス登録を行う（S93）。そして、ネイティブコードを実行し（S94）、S81の処理へ移行する。

【0070】また、前記S91の処理結果により、S94に移行してネイティブコードを実行し（S94）、S81の処理へ移行する。

【0071】㊦：例3の特徴

コンパイル処理は時間を要するため、一度しか実行はないようなバイトコードはコンパイルして実行するよりもインタプリタで実行する方が速い。そこで、前記のように処理を行うことで、アプリケーションプログラムの全体の実行速度を速くすることができる。また、前記例2と比較して、実行カウンタのための領域が必要なく、メモリを節約できる。

【0072】(4)：例4の詳細な説明

㊦：例4の内容

例4は、例1において、検索テーブルはコンパイルされているかどうかだけの情報（タグの情報）を持ち、ネイティブコードのアドレスはバイトコードのアドレスから計算する例である。

【0073】㊦：検索テーブルとコードキャッシュの構成例

図7は例4の検索テーブルとコードキャッシュの構成例である。この検索テーブル2には、タグ（tag）のみ格納されている。この場合、バイトコードアドレスの下位Nビットを利用して検索テーブル2をアクセスすると共に、前記Nビットの下位アドレスを演算器4により整数倍、例えば、8倍（ $\times 8$ ）した値によりコードキャッシュ3のネイティブコード領域のアドレスへアクセスできるようにになっている。

【0074】また、検索テーブル2は、バイトコードアドレス（上位Mビット+下位Nビットとする）の下位Nビットを利用してタグ（tag）を検索できるようになっている。そして、バイトコードアドレスの上位Mビットをコンパレータ1に入力すると共に、検索テーブルのタグ（tag）情報をコンパレータ1に入力し、両者が一致するか否かの比較を行う。前記比較の結果、両者が一致した場合、コンパレータ1から一致出力が出されるが、この一致出力信号によりコンパイル済みと判断する。

【0075】㊦：例4の特徴

バイトコードのアドレスに対し、コンパイルされるネイティブコードのメモリ上の位置が一意に決まるため、テーブルからネイティブコードのアドレスを索引する必要がなくなる。また、置換の際には、破棄すべきネイティ

ブコードを探す必要がなく単純に処理できる。

【0076】(5)：例5の詳細な説明

㊦：例5の内容

例5は、例1において、コンパイル時間を短縮（応答性を向上）するために、コンパイルする範囲を制限する。処理の途中でであっても、バイトコードの特定の命令（分岐命令、メソッド呼び出しなど）を検出したら、そこまでをコンパイルし、処理を中断する命令（ネイティブコードの分岐命令や割り込み命令）を生成し（ネイティブコードの最終コードの後に、分岐命令や割り込み命令を生成し）、残りはコンパイルしない例である。

【0077】すなわち、仮想計算機において、コンパイル時間を短縮するために、コンパイルする範囲を制限し、処理の途中でであっても、バイトコードの特定の命令を検出したら、そこまでをコンパイルして処理を中断する命令を生成し、残りはコンパイルしないように処理を制限する処理制限手段を備えている。

【0078】㊦：例5の特徴

必ず実行する部分だけがコンパイルされ効率が良い。例えば、“IF条件THEN処理”のようなバイトコードの列をコンパイルするとき、「条件」を処理する部分だけコンパイルを行い、「処理」の部分はコンパイルしない。

【0079】この場合、従来は、メソッド全体をコンパイルしていたので、全く実行しない部分もコンパイルしていた。これに対して例5では、必ず実行する部分だけをコンパイルするので、コンパイルに消費する時間の無駄がなく高速である。

【0080】また、従来は、メソッド全体をコンパイルしていたので、全く実行しない部分のネイティブコードも生成していた。しかし、例5では、必ず実行する部分だけをコンパイルするので、生成するネイティブコードに無駄がなく、メモリの消費が抑えられる。また、アプリケーションプログラムの応答性を向上させることができる。すなわち、コンパイルしている最中はアプリケーションプログラムの実行は停止するので、この停止している時間が短縮する。

【0081】(6)：例6の詳細な説明

㊦：例6の内容

例6は、コンパイル時間の短縮を行う例であり、例5において、特定な命令の検出のほかに、バイトコードの量でコンパイルを中断する。例えば、100命令コンパイルしたところでコンパイルを中断する例である。なお、例6は、例4と同様に、例1の処理におけるコンパイルを中断させる条件を提供するものである。

【0082】㊦：例6の特徴

例5と同じように、必ず実行する部分だけをコンパイルするので、生成するネイティブコードに無駄がなく、メモリの消費が抑えられる。また、アプリケーションプログラムの応答性を向上させることができる。すなわち、

コンパイルしている最中はアプリケーションプログラムの実行は停止するので、この停止している時間が短縮する。

【0083】(7)：例7の詳細な説明

①：例7の内容

例7はコンパイル時間の短縮を行う例であり、例5において、特定な命令の検出のほかに、生成したネイティブコードの量でコンパイルを中断する。例えば、コンパイルが128ワード分だけネイティブコードを生成したら、コンパイルを中断する。なお、例7は、例4と同様に、例1の処理におけるコンパイルを中断させる条件を提供するものである。

【0084】②：例7の特徴

例5と同じように、必ず実行する部分だけをコンパイルするので、生成するネイティブコードに無駄がなく、メモリの消費が抑えられる。また、アプリケーションプログラムの応答性を向上させることができる。すなわち、コンパイルしている最中はアプリケーションプログラムの実行は停止するので、この停止している時間が短縮する。

【0085】(8)：例8の詳細な説明

①：例8の内容

例8はコンパイル時間の短縮を行う例であり、特定な命令の検出のほかに、時間によってコンパイルを中断する。例えば、コンパイラがコンパイルの処理に100ミリ秒消費したら、コンパイルを中断する。なお、例8は、例4と同様に、例1の処理におけるコンパイルを中断させる条件を提供するものである。

【0086】②：例8の特徴

例5と同じように、必ず実行する部分だけをコンパイルするので、生成するネイティブコードに無駄がなく、メモリの消費が抑えられる。また、アプリケーションプログラムの応答性を向上させることができる。すなわち、コンパイルしている最中はアプリケーションプログラムの実行は停止するので、この停止している時間が短縮する。

【0087】(9)：例9の詳細な説明

①：例9の内容

例9は、コードキャッシュの置換を行う例であり、例1において、コードキャッシュに空きがなくなったとき、コードキャッシュの先頭から順にネイティブコードを破棄する。この処理では、FIFO (First In Last Out) 方式で最も昔にコンパイルしたコードから順に破棄する。

【0088】②：コードキャッシュの置換の説明

図8は例9のコードキャッシュの置換処理説明図である。この例では、初期値として、P1：命令生成ポインタはコードキャッシュの先頭、P2：解放ポインタはコードキャッシュの末尾を指しておく。実行が進む（コンパイルが何度も行われる）につれて、コードキャッシュ

3は小さなブロックに分けられ、ブロックの先頭にタグ (tag) ポインタ (検索テーブルへのポインタ) と、ネクスト (next) ポインタ (次のブロックを指すポインタ) が格納される。

【0089】ブロックの残り部分はネイティブコードを格納する命令領域である。ネクスト (next) ポインタはポインタである必要はなく、ブロックの切れ目が判れば良い。

【0090】③：フローチャートによる処理の説明

図9は、例9のコードキャッシュ置換処理フローチャートである。以下、図9に基づいて、例9の処理を説明する。なお、S101～S111は各処理ステップを示す。

【0091】この処理は、コンパイルの処理の最中にコードキャッシュ3の置換処理を行う例である。まず、生成する領域のタグ (tag) ポインタに検索テーブル2へのタグ (tag) ポインタを設定する (S101)。設定する領域は図8のP1 (命令生成ポインタ) によって指示される。S102、S103、S104、S105がコンパイル処理のメインループであり、バイトコードからネイティブコードを生成する処理である。

【0092】次に、JITコンパイルのコンパイル処理により、バイトコードからネイティブコードへの変換を行う (S102)。変換後のネイティブコードをコードキャッシュ3に格納する前に、命令生成ポインタと解放ポインタの値を比較 (命令生成ポインタ < 解放ポインタの条件を満たしているか否かを比較) する (S103) ことで、有効なコードを上書きしてしまわないかチェックする。

【0093】その結果、命令生成ポインタが解放ポインタより小さければ、全て解放済みの領域なので、コードキャッシュ3にネイティブコードを生成して書き込む (S104)。この時、命令生成ポインタを進める。次に、コンパイル終了条件を満たしたか否かを判断する (S105)。

【0094】その結果、コンパイル終了条件を満たさなければ、S102の処理へ移行し、コンパイル処理を継続する。また、コンパイル終了条件を満たしたら、生成したネイティブコードを1つのブロックとして構成する。この場合、生成した領域のネクストポインタに、命令生成ポインタの値を設定する (S106)。

【0095】また、S103の処理で、前記条件を満たしていない場合、すなわち、命令生成ポインタが解放ポインタと等しいか、又は大きい場合、有効なブロックを解放する必要がある。そこで、解放ポインタがコードキャッシュの最後を指している (解放ポインタ < コードキャッシュの末尾の条件を満たしている) か否かをチェックする (S107)。

【0096】その結果、解放ポインタがコードキャッシュの最後を指している場合は、解放ポインタをコードキ

キャッシュ3の先頭に設定し(S108)、命令生成ポイントをコードキャッシュ3の先頭のブロックの命令領域に設定する(S109)。この処理により、コードキャッシュ3の先頭に戻ってブロックの解放を行う。

【0097】次に、解放ポイントが示すタグ(tag)ポイントから検索テーブル2のタグ(tag)をクリアする(S110)。そして、前記S110の処理により、検索テーブルのタグ(tag)が無効化され、解放ポイントが示すブロックが実際に解放される。次に、解放ポイントを次のポイント(nextポイント)に設定して(S111)、S104の処理へ移行する。この処理で、解放ポイントが次のブロックを指すようになる。

【0098】④：特徴

コードキャッシュから検索テーブルへのポイントを持つことによって、キャッシュブロックのサイズを可変長にでき、メモリを有効に活用することが可能になる。このポイントがない場合でも、検索テーブル全体を調べてキャッシュを無効化することができるが、検索テーブル全体を調べるのは処理量が多く、コンパイルに時間を浪費するが、例9の処理により、キャッシュの無効化を高速に行うことができる。

【0099】(10)：例10の詳細な説明

①：例10の内容

図10は例10のコードキャッシュの置換処理説明図である。例10は、コードキャッシュの置換を行う例であり、コードキャッシュ3に空きがなくなったとき、検索テーブル2の実行カウンタを値の小さいものをスキャンし、実行カウンタの値の小さなものから順に破棄する。実行カウンタはコンパイル直後は適当な大きな値を設定しておき、実行する度に実行したコードに対応するカウンタを増加するようにし、置換が必要となったときに、全体のカウンタを減少するようにする。

【0100】②：フローチャートによる処理の説明

図11は例10の処理フローチャートである。以下、図11に基づいて例10の処理を説明する。なお、S121～S136は各処理ステップを示す。また、S121～S131の処理は、図5に示した例2のS61～S75の処理と同じである。

【0101】インタプリタの処理では、コンパイル済みか否かを判断し(S121)、コンパイル済みでなければ、テーブル登録済み(図4の検索テーブル2のtagに登録されているか否か)を判断する(S122)。その結果、テーブル登録済みでなければ、タグ(tag)にアドレス登録を行い(S123)、実行カウンタをクリアする(S124)。

【0102】次に、命令フェッチを行い(S125)、デコードを行って(S126)、前記処理と同様にして命令を実行する(S127)。そして、S121の処理に移行する。

【0103】また、前記S122の処理でテーブル登録

済みであると判断した場合には、検索テーブル2の実行カウンタをUP(+1)し(S129)、予め決めた特定回数より大きいかなかを判断する(S130)。その結果、特定回数より小さければ、S125に移行しインタプリタで実行し、特定回数以上の場合は、JITコンパイラによりバイトコードをコンパイルしてネイティブコードを生成し(S131)、生成したネイティブコードをコードキャッシュ3に登録する(S132)。

【0104】また、この時、検索テーブル2の実行カウンタを設定する。次に、前記S132の処理で登録したネイティブコードを実行し(S133)、S121の処理へ移行する。また、S121の処理で、コンパイル済みであると判断した場合は、検索テーブル2の実行カウンタをUP(+1)して(S128)、S133の処理へ移行する。

【0105】一方、前記S131の処理において、ネイティブコードを生成した場合、コードキャッシュ3に空きがあるか否かを判断し(S134)、空きがあれば、S131の処理へ戻る。しかし、コードキャッシュ3に空きがなければ、実行カウンタ全体をスキャンし、個々の実行カウンタを減少させる(S135)。そして、実行回数が少ないエントリを破棄して、コードキャッシュ3に空きを作成し(S136)、S131の処理に戻る。

【0106】③：例10の特徴

頻繁に実行されるネイティブコードが破棄されにくくなる。しかし、空き領域が断片化し、ネイティブコード上で断片化した箇所への無駄な分岐命令が必要となる。

【0107】(11)：例11の詳細な説明

①：例11の内容

例11は、コードキャッシュ3のコンパクションの例であり、例10において、コードキャッシュ3の空き領域の断片化が起きる。この時に、ネイティブコードのかたまりを移動(リロケーション)することで、コンパイラが一度に生成するコードを一塊にまとめる。

【0108】②：例11の特徴

断片化により分断された命令の間に無駄な分岐命令を挿入する必要がなくなる。実行する命令列がコードキャッシュ上に分断されることがなくなり、一塊となるため、命令キャッシュが効率的に使われ、性能が向上する。

【0109】(12)：例12の詳細な説明

①：例12の内容

例12は、検索テーブルのヒット率の向上を図る例であり、例5において、コンパイルした全てのバイトコードの命令を検索テーブルに登録せず、コンパイルした命令シーケンスの先頭だけ検索テーブルに登録する。

【0110】②：例12の特徴

検索テーブルのエントリ数にも制限があり、コンパイルした全てのバイトコード命令のアドレスを登録すると、検索テーブル2を上書きする機会が増えるため、コンパ

ーションプログラムの全体の実行速度を速くすることができる。また、前記(2)と比較して、実行カウンタのための領域が必要なく、メモリを節約できる。

【0128】(4)：請求項4では、処理制限手段は、コンパイル時間を短縮するために、コンパイルする範囲を制限し、処理の途中でであっても、バイトコードの特定の命令を検出したら、そこまでをコンパイルして処理を中断する命令を生成し、残りはコンパイルしないように処理を制限する。

【0129】この場合、従来は、メソッド全体をコンパイルしていたので、全く実行しない部分もコンパイルしていた。これに対して請求項4では、必ず実行する部分だけをコンパイルするので、コンパイルに消費する時間の無駄がなく高速である。

【0130】また、従来は、メソッド全体をコンパイルしていたので、全く実行しない部分のネイティブコードも生成していた。しかし、請求項4では、必ず実行する部分だけをコンパイルするので、生成するネイティブコードに無駄がなく、メモリの消費が抑えられる。また、アプリケーションプログラムの応答性を向上させることができる。すなわち、コンパイルしている最中はアプリケーションプログラムの実行は停止するので、この停止している時間が短縮する。

【0131】(5)：請求項5では、第1の破棄手段は、コードキャッシュに空きがなくなった時、コードキャッシュの先頭から順にネイティブコードを破棄する。そして、第2の破棄手段は、FIFO方式で最も昔にコンパイルしたコードから順に破棄する。このようにすれば、ネイティブコードの破棄の方式が簡単のため、実現し易く、ネイティブコードの断片化が起きない。

【図面の簡単な説明】

【図1】本発明の実施の形態における例1の処理フローチャート（その1）である。

【図2】本発明の実施の形態における例1の処理フローチャート（その2）である。

【図3】本発明の実施の形態における例1の検索テーブル例である。

【図4】本発明の実施の形態における例2の検索テーブル例である。

【図5】本発明の実施の形態における例2のコンパイル選択処理フローチャートである。

【図6】本発明の実施の形態における例3のコンパイル選択処理フローチャートである。

【図7】本発明の実施の形態における例4の検索テーブルとコードキャッシュの構成例である。

【図8】本発明の実施の形態における例9のコードキャッシュの置換処理説明図である。

【図9】本発明の実施の形態における例9のコードキャッシュの置換処理フローチャートである。

【図10】本発明の実施の形態における例10のコードキャッシュの置換処理説明図である。

【図11】本発明の実施の形態における例10の処理フローチャートである。

【図12】従来のインタプリタの処理フローチャートである。

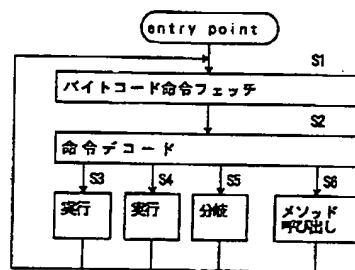
【図13】従来のインタプリタとJITコンパイラを組み合わせた時の処理フローチャートである。

【符号の説明】

- 1 コンパイラ
- 2 検索テーブル
- 3 コードキャッシュ
- 4 演算器

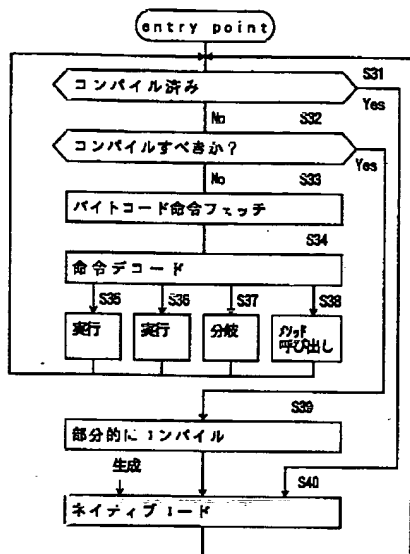
【図12】

インタプリタの処理フローチャート



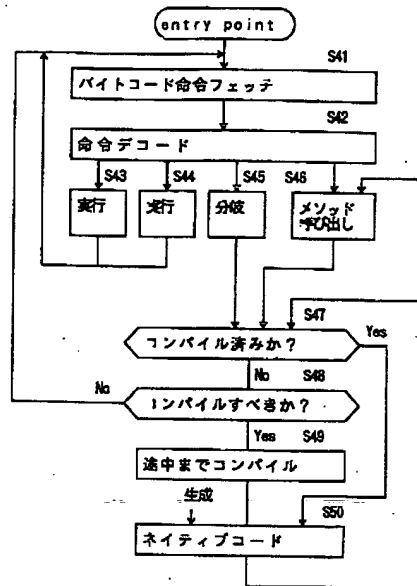
【図1】

例1の処理フローチャート（その1）



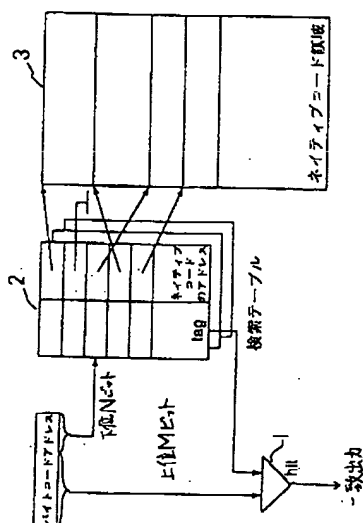
【図2】

例1の処理フローチャート(その2)



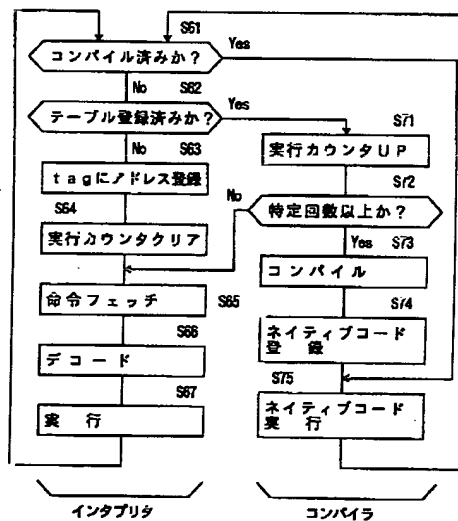
【図3】

例 1 の検索テーブル例



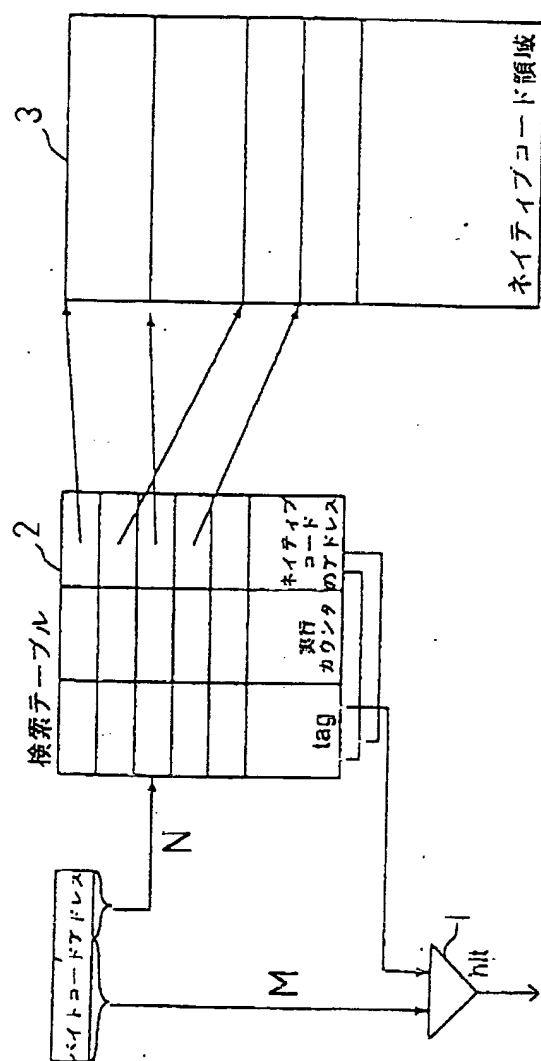
【図5】

例2のコンパイル選択処理フローチャート



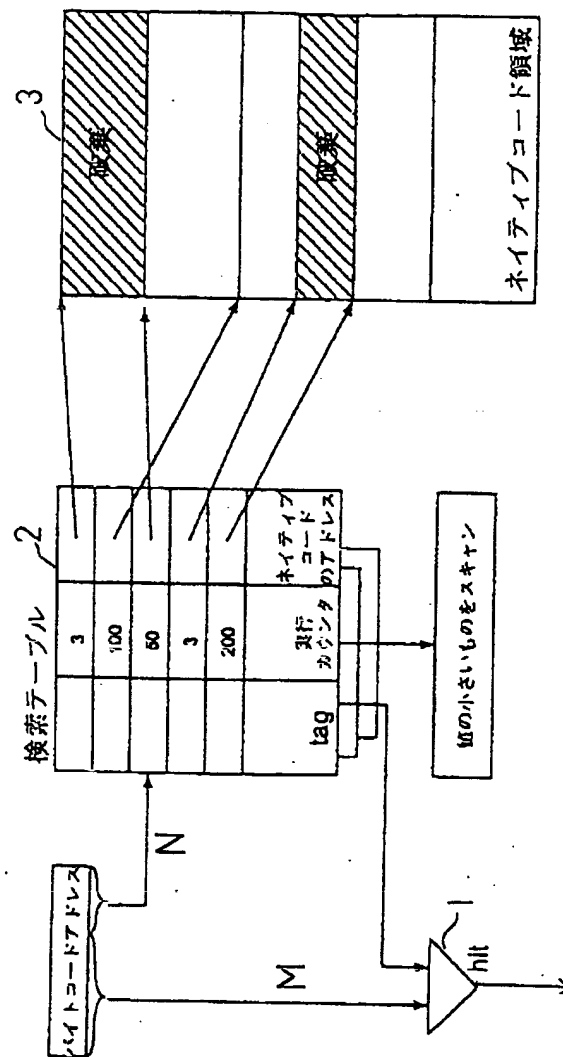
【図4】

例2の検索テーブル例



【図10】

例10のコードキャッシュの置換処理説明図



【図11】

例10の処理フローチャート

